System Design Mock Interview: Design Facebook Messenger

Key Product Goals

🎯 Functional Goals:

1. **Real-Time Messaging** → Users should be able to send and receive messages with minimal delay.
2. **Group Messaging** → Multiple users should be able to communicate within a shared conversation.
3. **Online Presence** → The system should display users' online/offline status.
4. **Media Sharing** → Support for images, videos, and files in messages.
5. **Read Receipts & Notifications** → Users should know when their messages are seen.
6. **Cross-Device Synchronization** → Messages should be available across multiple devices.
7. **Search & History** → Users should be able to search and retrieve old messages.
8. **Security & Privacy** → Messages must be encrypted and protected from unauthorized access.

🏗️ Technical Goals & Constraints:

- **Scalability** → The system must handle millions of users with high throughput.
- **Low Latency** → Messages should be delivered with minimal delay.
- **Fault Tolerance** → The system must continue operating even if some servers fail.
- **Data Consistency vs. Availability** → Prioritizing high availability over strict consistency.
- **Efficient Storage & Retrieval** → Storing billions of messages efficiently.

🔗 Networking Challenges & Solutions

Since direct **peer-to-peer** messaging is impractical, a **centralized chat server** is required to relay messages. Several approaches are evaluated:

❌ 1. HTTP Polling (Inefficient)

- The client continuously asks the server for new messages.
- Wastes bandwidth and causes high latency.

⚠️ 2. Long Polling (Better, but not Ideal)

- The server holds the request open until a new message arrives.
- Reduces unnecessary requests but still requires re-establishing connections.

✅ 3. WebSockets (Recommended Solution)

- Maintains a **persistent** full-duplex connection.
- Allows **instant message delivery** and reduces overhead.

⚖️ Scaling WebSockets

WebSockets require **persistent connections**, creating **scalability challenges**:

- A single server can only handle **~65,000 TCP connections** due to port limitations.
- **Solution**: Use **multiple API servers** managed by a **load balancer** to distribute WebSocket connections.

🏗️ Load Balancer Strategy

1. **Distributes WebSocket connections** to different API servers.
2. **Prevents overload** on a single server.
3. **Routes users to the nearest or least-busy server.**

📡 Message Routing & Queueing

With multiple API servers, a **Pub/Sub Messaging Queue System** ensures efficient message distribution:

✅ How Pub/Sub Works

1. **Messages are sent to a central queue** instead of directly to a recipient.
2. **API servers subscribe** to user-specific message updates.
3. **The queue ensures messages are delivered** reliably even if servers fail.
4. **Scales horizontally** to support **millions of concurrent messages**.

📩 Offline Message Handling

- If a user is offline, their messages are stored in a **message queue** and delivered when they reconnect.

🗄️ Database & Storage

📌 Choosing the Right Database

A **NoSQL database** is preferred for scalability:

- **Cassandra / HBase** → Supports **horizontal scaling & automatic sharding**.
- **CAP Theorem Trade-offs** → Prioritizing **Availability & Partition Tolerance (AP)** over strict **Consistency (C)** since minor delays in message ordering are acceptable.

🏗️ Database Schema

The database consists of the following tables:

1. **Users Table** → Stores user ID, username, and last active timestamp.
2. **Messages Table** → Contains message ID, sender, recipient, timestamp, and message content.
3. **Conversations Table** → Stores group chat information.
4. **User-Conversation Mapping Table** → Tracks which users belong to which conversations.

💾 Caching & Performance Optimization

To improve performance and reduce database queries:

1. **Redis / Memcached** → Used as a caching layer to store recently accessed messages.
2. **Read-Through Cache** → Ensures frequently accessed messages are fetched from cache before querying the database.
3. **Content Delivery Network (CDN)** → Used to distribute **images & videos** efficiently.

📌 CDN Strategy for Media Files

- Messages with images/videos **store only the file URL** in the database.
- Users retrieve media from **Amazon S3 / Google Cloud Storage** instead of loading it from the backend.
- **CDN caching** accelerates image/video loading and reduces latency.

📩 Notification System

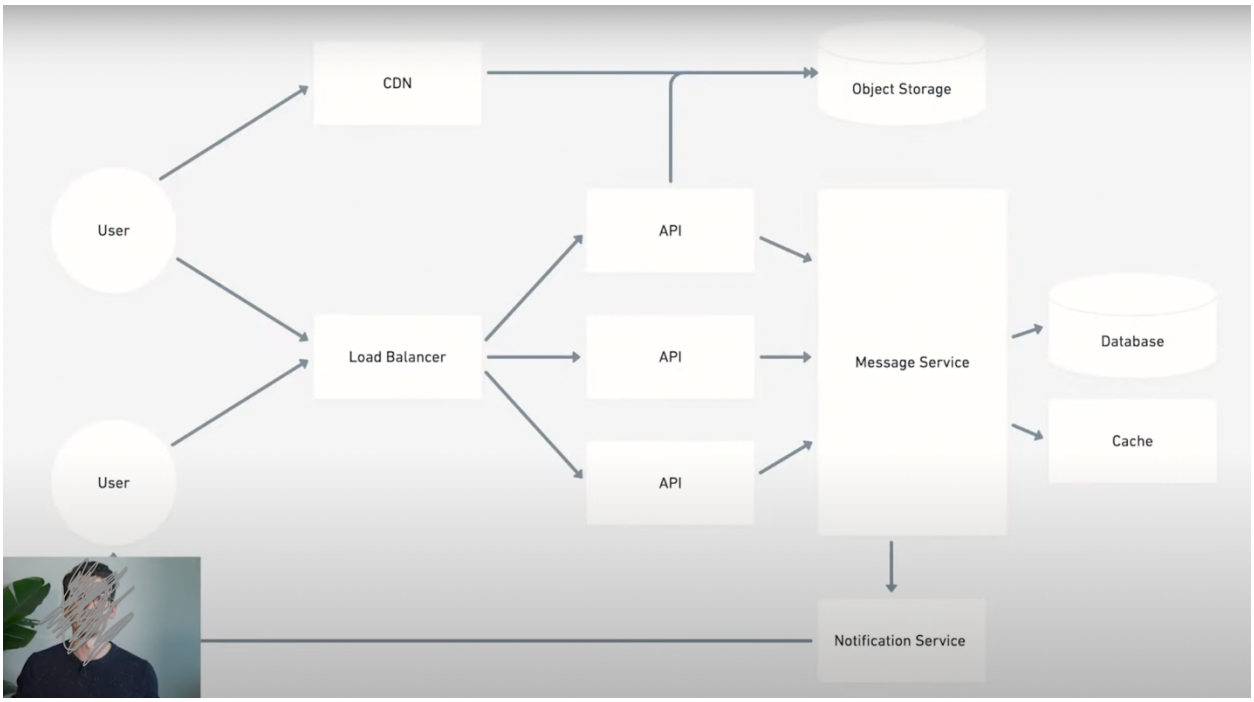
Offline users need to be **notified** about new messages:

- A **notification service** triggers alerts when a message is received.
- Integrates with **Push Notification APIs**:
  - **Apple Push Notification Service (APNS) for iOS**
  - **Firebase Cloud Messaging (FCM) for Android**
  - **Email/SMS notifications** for fallback.

🏗️ Final Architecture Overview

The system integrates **multiple components**:

1. **Load Balancer** → Distributes WebSocket connections.
2. **API Servers** → Handle user authentication and WebSocket connections.
3. **Message Queue (Pub/Sub)** → Routes messages efficiently.
4. **NoSQL Database** → Stores messages and user metadata.
5. **Caching Layer** → Reduces database calls.
6. **Object Storage + CDN** → Optimizes media file access.
7. **Notification Service** → Alerts offline users about new messages.



Answer

users	
id	int
username	string
lastActive	timestamp

messages	
id	int
user	int
conversation	int
text	string
media_url	string

conversations	
id	int
name	string

conversation_users	
conversation	int
user	int

🔢 Insights Based on Numbers

1. **65,000 TCP connections per server** → Requires horizontal scaling.
2. **Millions of messages per second** → Demands a high-throughput system.
3. **Thousands of API servers** → Needed to balance WebSocket connections and ensure performance.

4. **Latency under 100ms** → Crucial for a seamless user experience.

YT Video: https://www.youtube.com/watch?v=uzeJb7ZjoQ4

My chatgpt: https://chatgpt.com/g/g-GvcYCKPIH-video-summarizer/c/67bd3552-0804-800f-b76d-b83c80094f48